**Exercise 7: Financial Forecasting**

### ****1. Understand Recursive Algorithms****

**Recursion** is a technique where a function calls itself to solve a smaller instance of the same problem. It’s often used when a problem can be divided into subproblems of the same type.

**Advantages:**

1.Cleaner, more elegant code for problems like factorial, Fibonacci, tree traversal.

2.Helps solve problems where iteration would be more complex.

**Disadvantages:**

1.Higher memory usage due to call stack.

2.Risk of stack overflow if not implemented properly.

### ****2. Setup: Recursive Future Value Function****

Let’s say we want to forecast a financial value (e.g., revenue or investment amount) over n years, given:

Initial amount -P

Annual growth rate -r(as a percentage)

The formula for future value is : FV=P.(1+r)^n

Using recursion: FV(n)=FV(n-1).(1+r)

with base case: FV(0)=P

### ****3. Analysis****

#### ****Time Complexity:****

**T(n) = T(n-1) + O(1)** → **O(n)**

One recursive call per year

#### ****Space Complexity:**** O(n) due to recursive call stack

### ****Optimizing the Recursive Solution****

## Problem in Original Recursive Code:

public static double futureValueRecursive(double P, double r, int n) {

if (n == 0) {

return P;

}

return futureValueRecursive(P, r, n - 1) \* (1 + r);

}

### Problem:

1.For every call to n, the function must compute n-1, then n-2, and so on — even if it already calculated those values before.

2.This leads to **repeated recalculations** and unnecessary growth of the **call stack**.

3.Example: Calling futureValueRecursive(P, r, 100000) could crash with StackOverflowError.

### How Memoization Helps:

We **store** the result of futureValue(P, r, i) for each i, so we never recompute it.

**Modified C# Code:**

using System;

using System.Collections.Generic;

class FinancialForecastMemo

{

static Dictionary<int, double> memo = new Dictionary<int, double>();

public static double PredictFutureValue(int year, double initialValue, double growthRate)

{

if (year == 0)

return initialValue;

if (memo.ContainsKey(year))

return memo[year];

double result = PredictFutureValue(year - 1, initialValue, growthRate) \* (1 + growthRate);

memo[year] = result;

return result;

}

static void Main(string[] args)

{

double initialValue = 1000;

double growthRate = 0.08;

int targetYear = 10;

double result = PredictFutureValue(targetYear, initialValue, growthRate);

Console.WriteLine($"Predicted value after {targetYear} years: {result:F2}");

}

}

**Benefits of This Optimization:**

| **Feature** | **Before (Pure Recursion)** | **After (Memoized)** |
| --- | --- | --- |
| Time Complexity | O(n) | O(n) |
| Space Complexity | O(n) stack | O(n) stack + O(n) cache |
| Redundant Computation | Yes | Eliminated |
| Speed | Slower | Much Faster |
| Max Safe Depth | Small (risk of overflow) | Larger (more stable) |